Chapter 1: Getting Started with JGrasp™

The tools and your first programs

This chapter introduces you to using JGrasp™ from the CD. You should only choose this option if you are comfortable with programming and setting up applications. Familiarity with using a version of **`make`** will also be an advantage.

It may be that you are familiar with the kind of tools used with a compiled language because one of the languages you already use has similar tools. If so, please excuse me for taking time to explain things for the benefit of other readers.

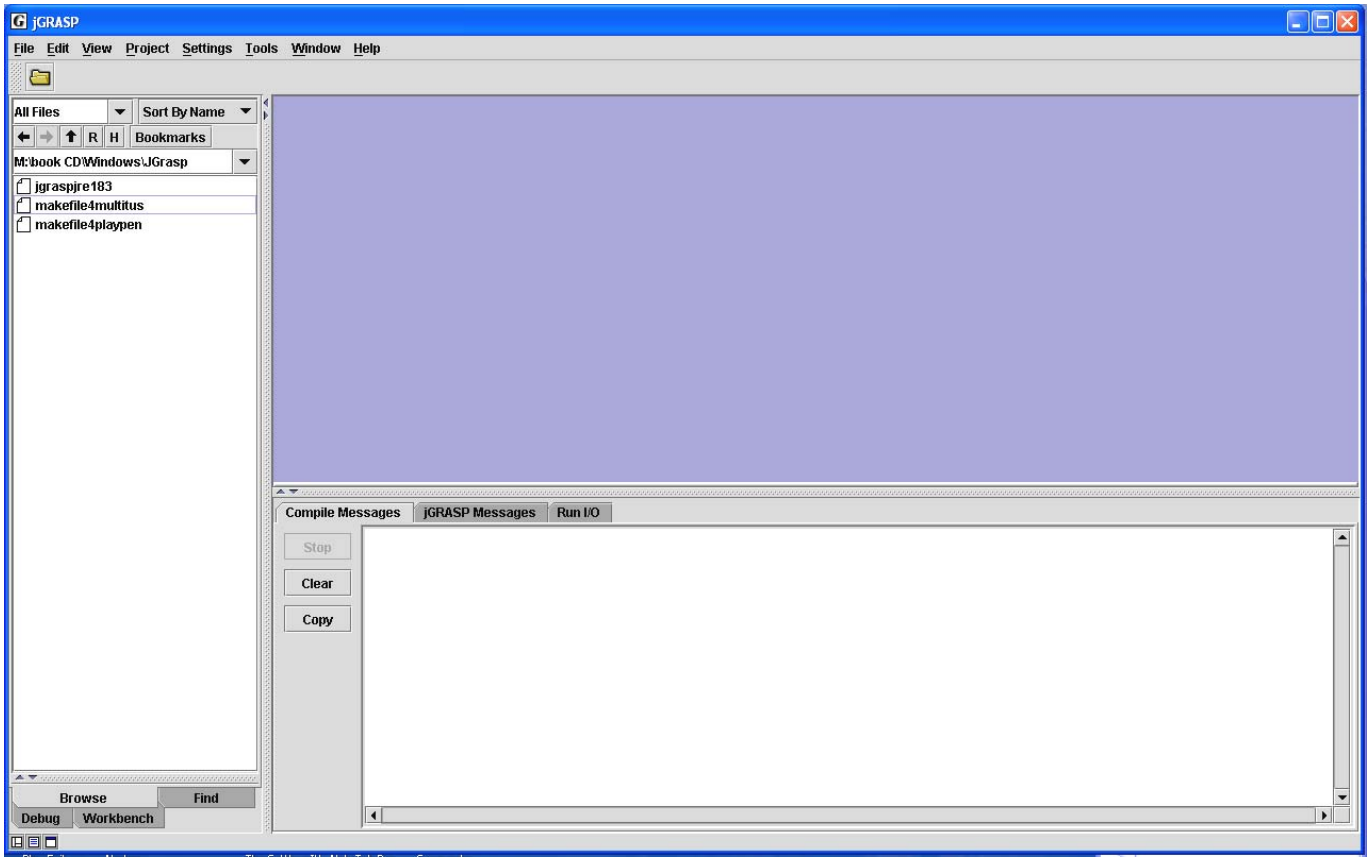
I am now going to walk through the process of producing three simple programs from scratch. The first program is the traditional 'Hello World'. The second program introduces you to the specifics of using a `makefile` for projects that have multiple files of source code. The third program checks that everything has been correctly set up to use my graphics library. These programs are deliberately simple so that we can focus on the process of creating a program from source code and libraries. As I go, I will add information that may be new to you if you are used to a language that is substantially different from C++.

I assume that you have followed the instructions for setting up JGrasp™ on your system and that you have set it up to use the compiler of your choice. When you use my library, you will also need to compile it from the source code using the `makefile` provided for that purpose. If you are not using Gnu **`make`**, you may have to edit the `makefile`.

Creating a 'Hello World' Program

The first step is to launch the IDE. I always have the IDE icon on my desktop. If you accepted that option when you installed the software from the CD, you will find this icon somewhere on your desktop:

If you chose otherwise, you will have to launch JGrasp™ differently. When you have launched JGrasp™, you should see something like this window on your screen:

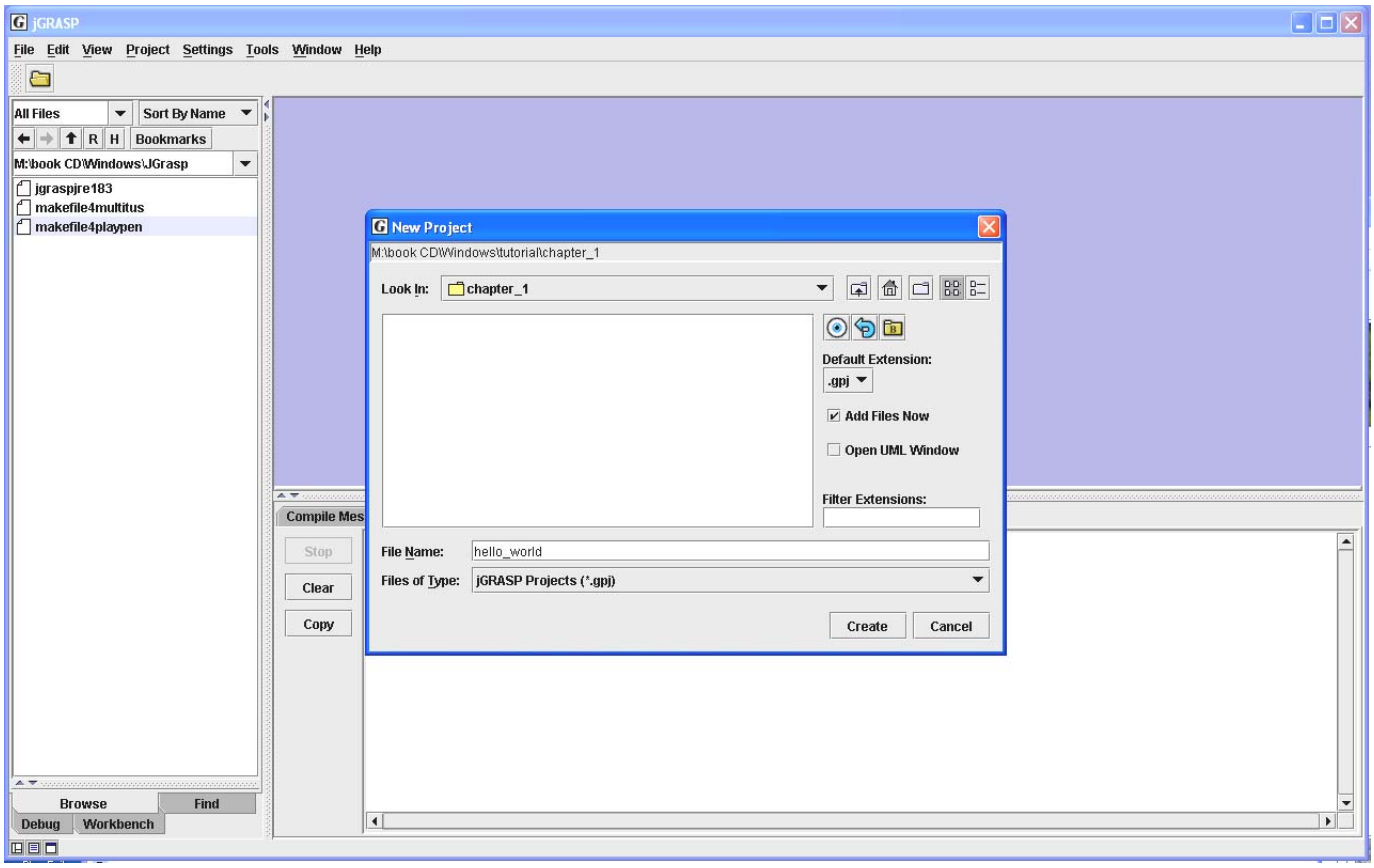


The exact details of the left hand pane will depend on how you installed JGrasp™ so do not worry about them. You will be able to navigate to where you need to be. You might find it useful to stop for a moment and play with the navigation controls at the top of the left hand pane. Note also that you can change between different views by using the tags at the bottom of the pane.

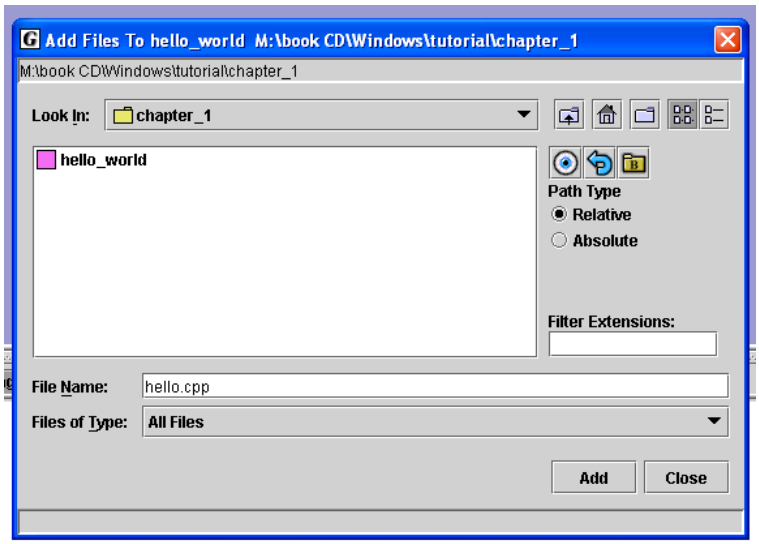
The various panes are adjustable and we will see what each is for as we go. If you are familiar with using an IDE, you will probably recognize most of what is on the screen. If you start pulling down menus, you will notice that most of the entries are 'grayed out'. In addition, most of the icons are 'grayed out' but, unless you have disabled the feature, you will get a tool-tip if you let the mouse cursor hover over an icon. These tips are minimal but I find the inclusion of the default hotkey in the tip a useful feature.

You may have noticed that starting up JGrasp™ also starts a console window. Please do not close the console window as it is what is underpinning the IDE.

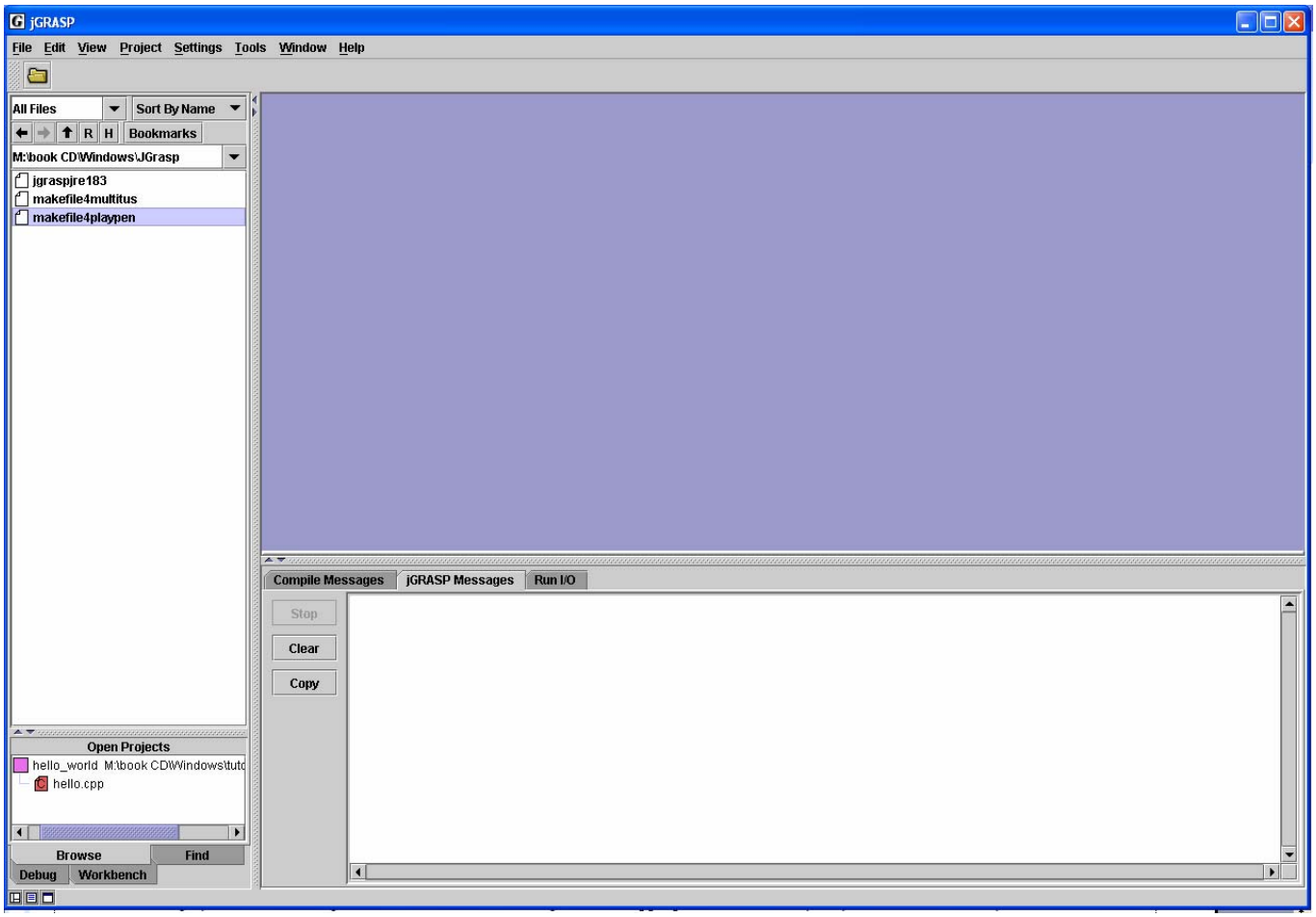
Your next step is to create a simple project. First, use the project menu to create a new project. You will need to set the 'look in' item to whereever you have placed your `tutorial\chapter_1` and then type in the project name. The result should be something like this:



Click on the create button and then enter `hello.cpp` as the file name to add to the project:

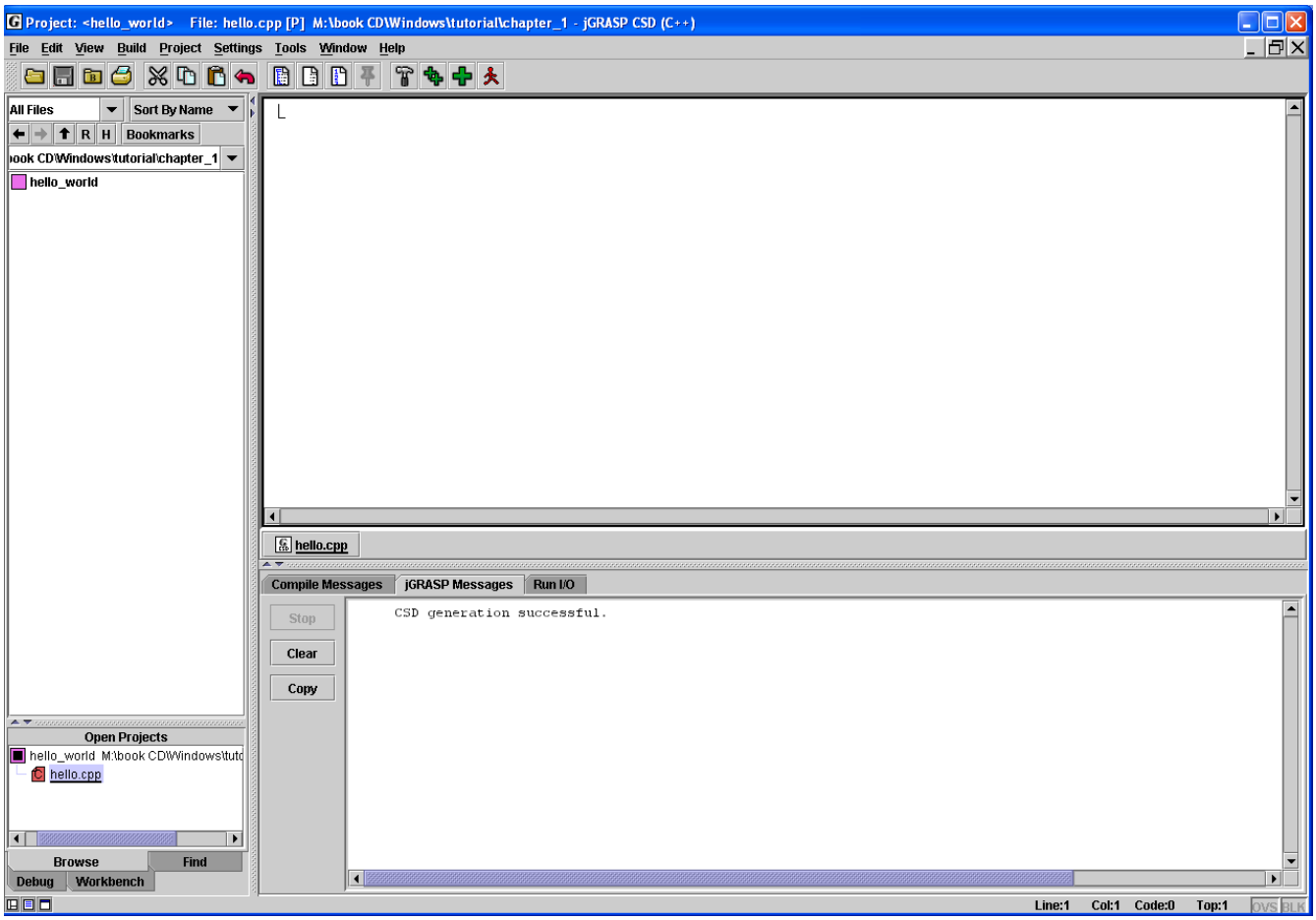


Click on the Add button and then the Close button. The result should be:



Note the change in the bottom left corner. Please navigate to the same place in the upper pane as that will make it easier to track what is happening. Note that there is not yet a `hello.cpp` file listed in the upper part of the browse pane. Select `hello.cpp` from the 'Open Projects' sub-pane and click on `create` in the resulting dialog box.

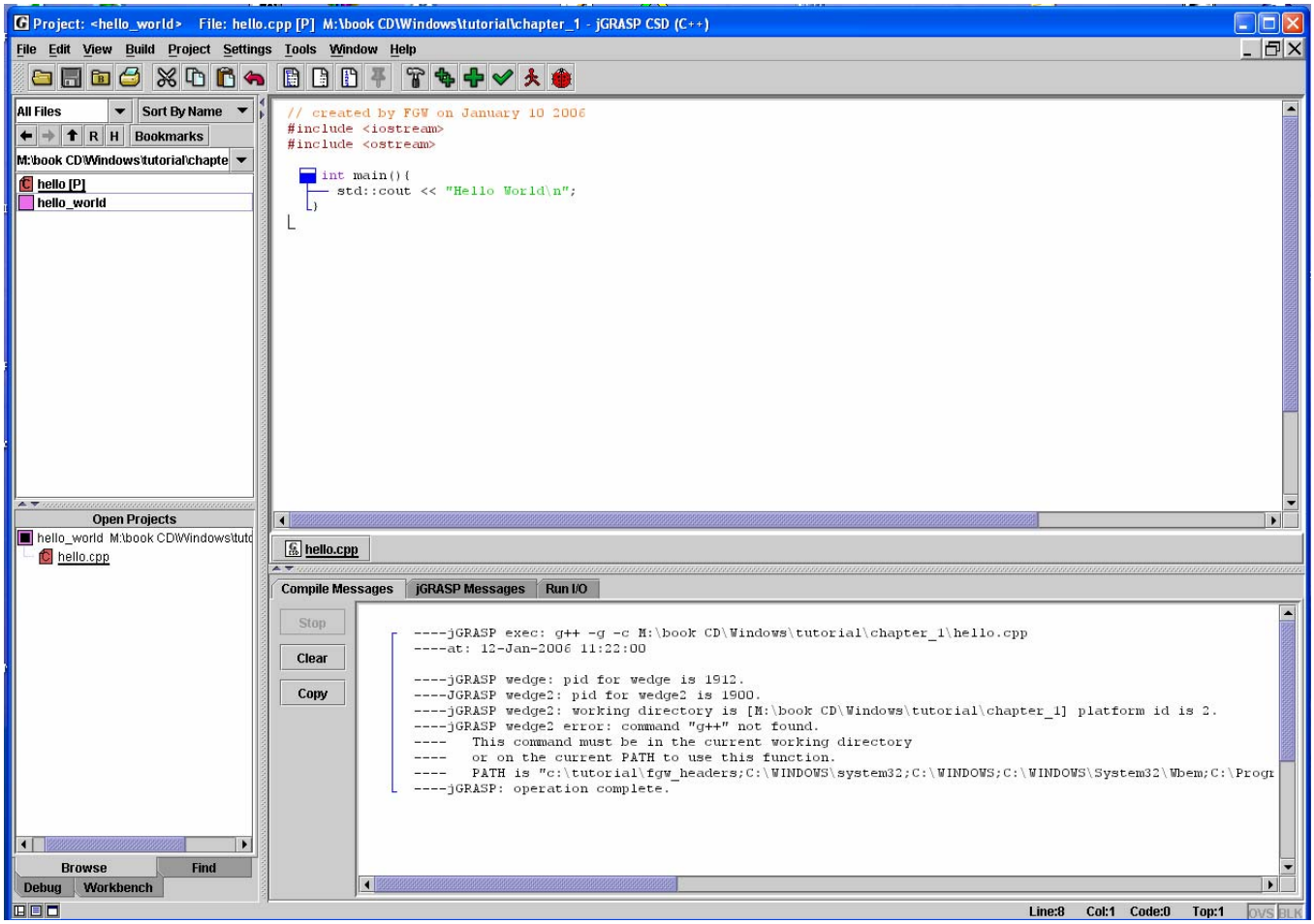
You should then see:



Next type the following short program into the upper right pane:

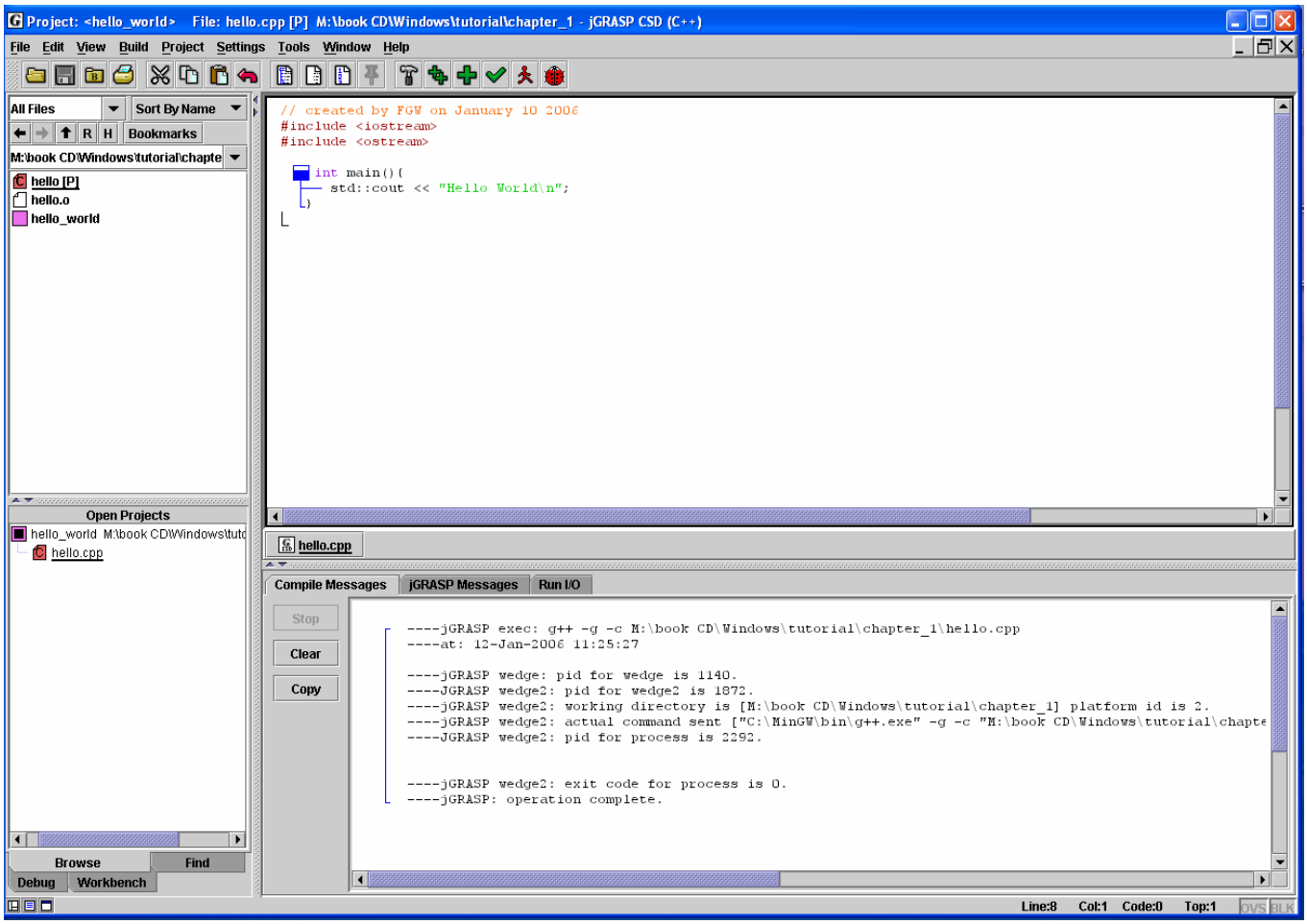
```
// created by FGW on January 10 2006
#include <iostream>
#include <ostream>

int main(){
 std::cout << "Hello World\n";
}
```

When you go to the build menu and select compile, one of several things may happen (sorry about that, but a lot depends on how you set up the material from the CD): it may compile successfully; it may complain that there is no compiler (in which case you need to use the settings menu to tell the IDE which compiler you intend to use); or it may produce something like:



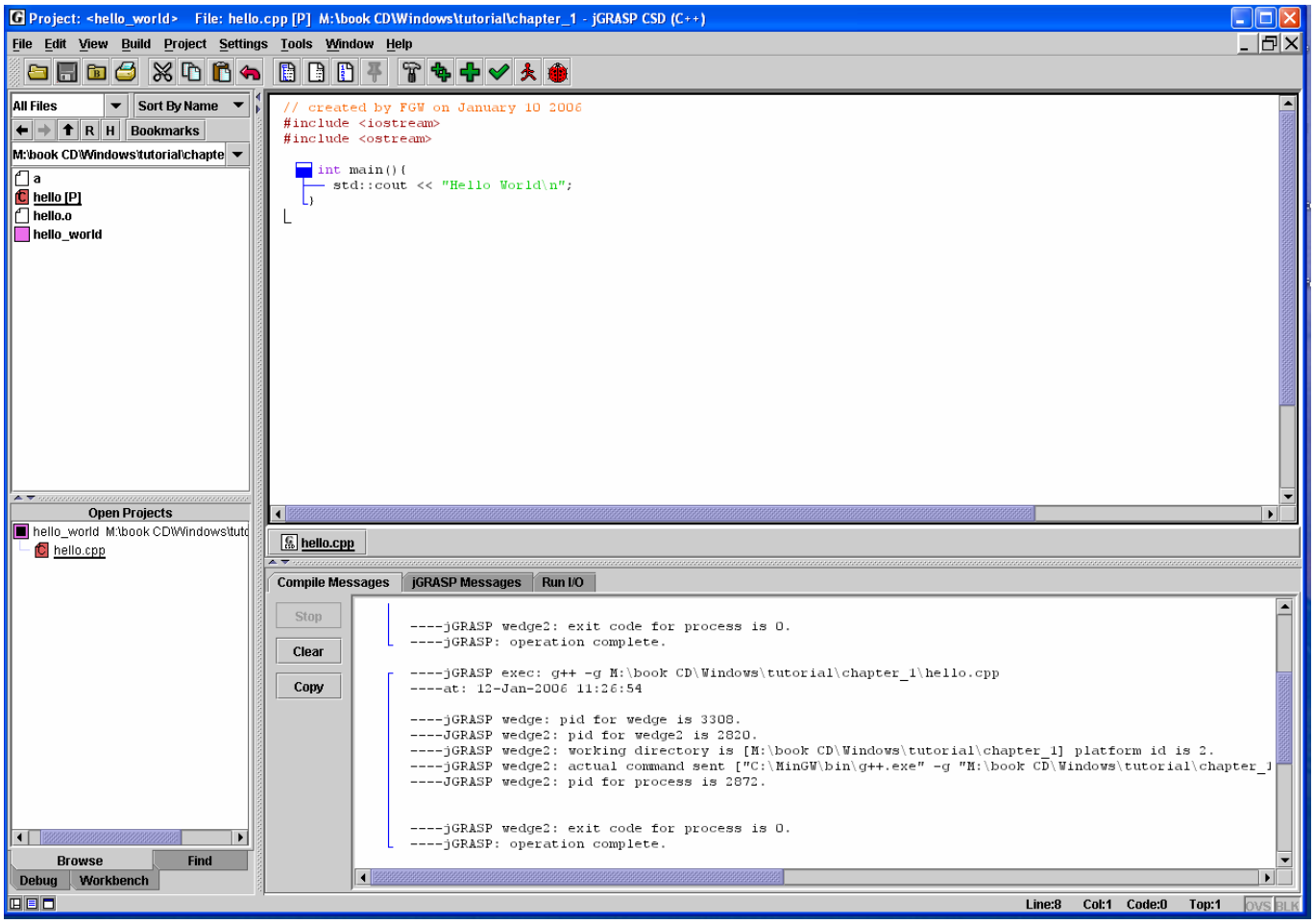
If you read the message in the lower pane, you will see that it is saying that it cannot find the compiler in the system path. If this is the case, you will need to edit the system path to include the folder or directory that contains the compiler. On Windows XP, you have to edit the environment variables. Go to 'control panel', 'system', 'advanced', select 'environment variables'. Now edit the path by appending a semi-colon to the current path and then the location of the compiler you are using (for example, on the computer I am using it is `C:\MinGW\bin`). Now you have to close JGrasp™ and then restart it so that it will pick up the new path. (Note that JGrasp™ uses a copy of the environment at the time it is started so that any time you make changes to the environment you will need to restart JGrasp™.)

Once JGrasp™ can find the compiler you should see something like:



Note that the top left pane now includes a `hello.o`. That is the file generated by compiling a single file of source code. We need to turn this into an executable.

In JGrasp™, working at the current simple level, we need to select '`Compile and Link`' from the Build menu. When you do that, your screen should look like:



We have another new file in the default directory. This one is listed as 'a' but is actually `a.exe`. Please note that JGrasp™ always creates an executable called `a.exe`. When you elect to run (from the Build menu) the current project, it will execute the program called `a.exe`. If you want to name the program something else for external use you will have to use your operating system's facility for renaming files.

If you have got this far you are in a position to write, compile, link and run a simple C++ program from within JGrasp™.

What the code means

Let us look briefly at the seven lines that make up the 'Hello World' program.

The first line is a comment. Whenever the compiler meets a ***//***, it ignores everything from there to the end of the line. Comments are for humans and sometimes for special code-analysis tools; they are not for compilers. In this case, the comment is almost redundant, and I have included it only as an example.

The second and third lines tell the compiler that the following code may use names from the part of the Standard C++ Library concerned with streaming data in and out through the console. We call the parts in the angle brackets 'headers'; they tell the compiler to get relevant information from wherever it is keeping such details. Different compilers may obtain the information in different ways. In practice, a header is usually a text file in one of the compiler's subdirectories. (If you are interested, you will find the corresponding `iostream` file in `MinGWStudio\MinGW\include\c++\3.4.2` but I doubt that it will make much sense to you yet.)

The blank line has no significance and is purely there to separate the introductory part of the source code from the rest. The next line (**`int main(){`**) must exist exactly once in every program. Effectively, it determines where a program starts. There are some variants that allow the provision of data at program start-up, but that is all.

The fifth line is the substance of this program. **`std::cout`** is the name of a console output object. In other words, it is the name we use to designate an object that represents the console on the computer where the program will run; that is usually a window on the monitor screen. The part of the name before the double colon (**`std`**) tells the compiler that we are dealing with a name from the C++ Standard Library.

Language Note: C programmers will recognize the '<<' as being a left shift operator. In the context of an output object or destination, C++ reuses that operator as a streaming operator to insert data into an output stream.

The text in quotation marks tells the compiler that you want this text displayed. We call such quoted text a string literal. The **`\n`** at the end is the way we tell the compiler that we want to go to a new output line after this data has been displayed. The final element of the statement is the semicolon. That ends the statement and is the C++ equivalent of a full stop in English. Try leaving it out and then attempting to compile the code; you will see the kind of error message that results.

The last line of the program is a simple closing brace to match the opening brace at the end of the line with **main** in it. In general, we refer to code between an opening and closing brace as a block. In this context, the code between the opening brace and the closing brace is the definition of this version of **main** and specifies what will happen when the program executes. We call a block that defines a function 'the function body'. So

```
{
 std::cout << "Hello World";
}
```

is the body of the function **main()**.

Multifile Projects

The current release of JGrasp™ relies on the user writing a suitable `makefile` for any project that either uses third-party libraries or has more than one file of source code. This is the reason that I do not advise people who are unfamiliar with make files to use JGrasp™ for learning C++ unless they have a supervisor. The biggest single problem is that there is no universal standard for a `makefile`. The exact format depends on which version of the **make** utility you are using and even which system you are working on.

You will find suitable example `makefiles` for use with G++ in the folder or directory for the operating system you are using. Note that there are two examples, one is for multifile projects and the other is for using third-party libraries. If you study the examples, you should be able to produce one for a multifile project using a third-party library. If you are using some other **make** utility you will have to adjust the `makefile` to the requirements of that tool. You will also need to ensure that the utility is called **make**. If it is **gmake**, **nmake** etc. you will need to rename it **make**. There are notes about doing this in *Installing Software from the CD* file in the Windows directory on the CD.

The easiest way to deal with the `makefile` is to load the appropriate example (`makefile4multitus` in the JGrasp™ subdirectory for your operating system) into JGrasp™ by finding it and clicking on it. Now edit it for the source code files in your project and then save it as `makefile` in the directory for the current project. However, I am getting ahead of things because we do not even have a project.

Before you start, close the existing project, close any open files and clear the lower right pane. Now, create a new project called `multifile` in the `chapter_1` folder and add files called `file1.cpp` and `file2.cpp` to it. You do this in exactly the same way that you created the `hello_world` project and added `hello.cpp` to it.

Now open `file1.cpp` and add:

```
// multiple source code file example created by FGW January 10 2006
#include <iostream>
#include <ostream>
#include "file2.h"

int main(){
 foo();
}
```

If you try to compile that you will get a message about `file2.h` not being found. We will worry about the details later but for now you need to know that `file2.h` is something called a ***header file*** (note, not just a header) and it is a file of source code that provides some information about another file of source code. By convention, we name header files with the same name as the corresponding `.cpp` file but with a `.h` extension.

Note that the names of header files are placed in inverted commas and not angle brackets. That tells the compiler that it must look for them in the places that it has been told may include header files. This always includes the project directory so the simplest place to put your own header files is in that directory along with any other files you write.

Use the File menu to create a new C++ file. Type in the following code (please do not try to fathom it out, we will get there eventually but for now just treat it as a necessary magic incantation):

```
#ifndef FILE2_H
#define FILE2_H
void foo();
#endif
```

Save the file as `file2.h`. Return to `file1.cpp` and try to compile it. Barring typos, it should compile. However if you try to link it you will get an error effectively saying that the linker cannot find **foo**. Open (and create) `file2.cpp` and insert the following code:

```
#include <iostream>
#include <ostream>
#include "file2.h"

void foo(){
 std::cout << "this program was created from two .cpp files\n";
}
```

You will now be able to compile this file. Note the growing list of files in the top left hand pane of JGrasp™. However you will not be able to create a program from `file1.cpp` and `file2.cpp` because JGrasp™ does not understand the mechanism needed to do this (do not be too critical of JGrasp™: it is trying to manage multiple languages as well as user-selected compilers; most IDEs stick with pre-determined compilers or a single language). We need to tell JGrasp™ how to create an executable from multiple `.cpp` files. This is the job of our `makefile` working with the **make** utility.

Find the file called `makefile4multitus` and load it into the JGrasp™ editor. Now find all the occurrences of `one` (`one.cpp` and `one.o`) and change them to `file1` (to give `file1.cpp` and `file1.o`). Next do the same for `two` changing it to `file2`. Save the result as `makefile` in the project directory (`chapter_1`). Select either of the `.cpp` files from the project and select **make** from the Build menu. Now you should be able to run this program.

The program is trivial but the purpose is to provide you with a guided tour of working with multiple source code files in JGrasp™ projects.

What the code means

`file1.cpp` is very similar to `hello.cpp` except that it includes a header file as well as the two C++ headers. Note the different syntax for including a header file. Getting the wrong version of the syntax (using angle brackets where quotes are required or vice versa is a common cause of problems with getting code to compile.)

The body of **`main()`** consists of a single statement that 'calls' a function, **`foo`**. At the time that `file1.cpp` is compiled the compiler does not have details of **`foo()`** but it has been told just enough about it in `file2.h` so that it can leave the details to the linker (whose job is to combine different files and libraries to create a complete program). As we do not use any of the Standard Library in `file1.cpp` we could (and probably should) have omitted the inclusion of the two headers. It is a good principle to restrict inclusions to what is necessary.

The first two lines and the last line of `file2.h` are a precaution to ensure that a header file is not accidentally included more than once. I will go into more detail in a later chapter. The guts of `file2.h` is the line:

```
void foo();
```

That line promises the compiler that the linker will have access to a function called **`foo()`**. Again, I leave the further details for later.

`file2.cpp` provides those extra details by giving a definition of **`foo()`**. In other words, it provides the body of **`foo()`**. This time the two Standard Library headers are required. Some programmers might omit the inclusion of the header file, `file2.h`, but in my experience, it is usually sensible to include the corresponding header file. It allows the compiler to do a little more checking and increases its ability to detect errors. In addition, there are times when the compiler requires that information because it has to carry out some checks. If you get into the habit of including header files into the files they describe, you will not have to worry about when it is necessary and when it is not.

The `makefile` tells the make utility what has to be done to create the program you want. (Makefiles can do many other things as well.)

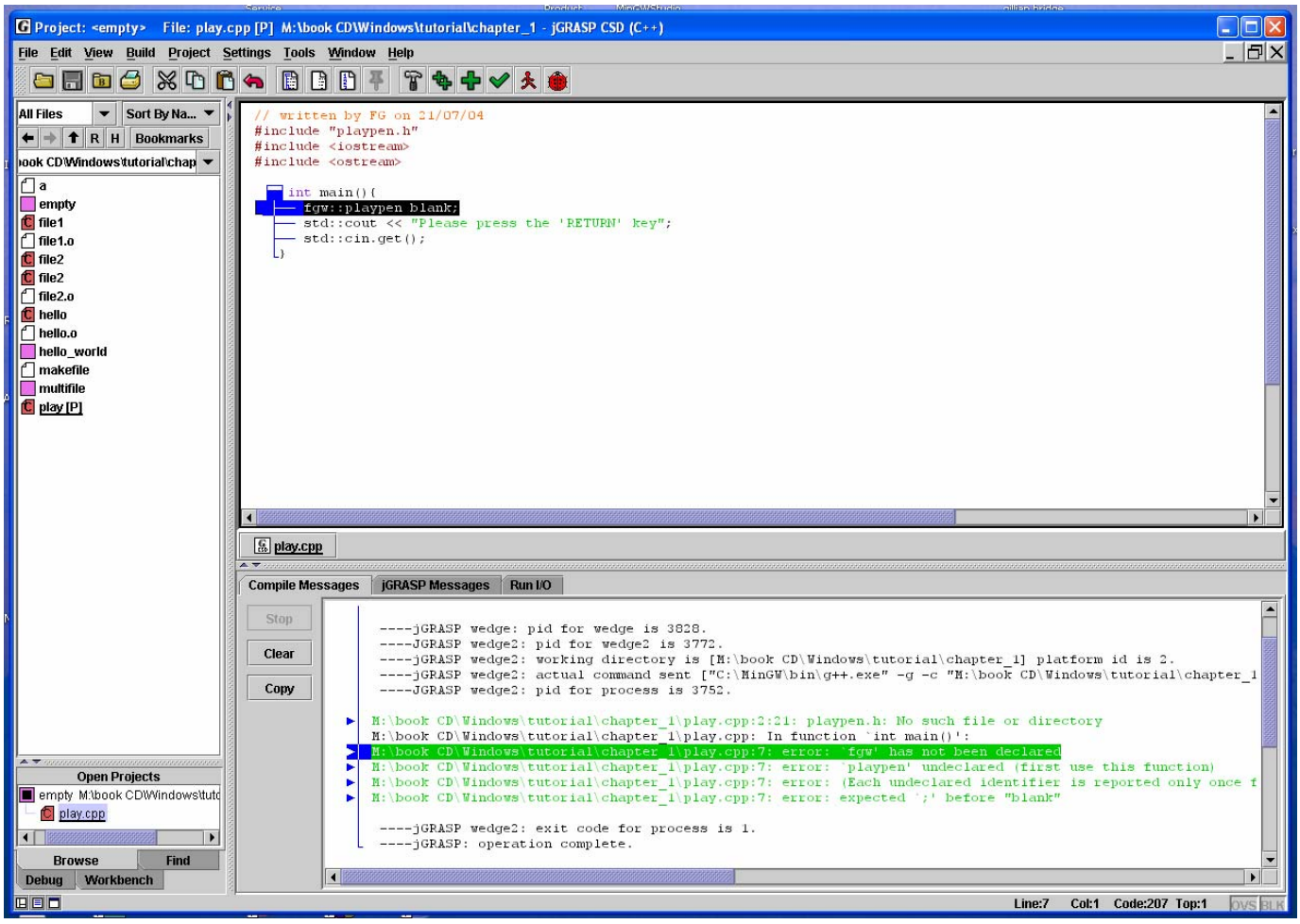
Using a Third-party Library – An Empty Playpen

From time to time, we are going to use a very special graphics window that I designed and several of my colleagues helped to implement. It is called the Playpen (as you use it I think you will come to appreciate the choice of name). This is a fixed size (512 by 512) graphics window with each pixel limited to one of 256 colors. Modern computers are usually capable of displaying many more colors than that but I wanted something that was very portable as well as something that would allow my readers and students to learn about simple graphics systems. For now we are going to use the Playpen with its default palette. (Later we will discover that we can choose different sets of 256 colors.)

Start up JGrasp™ and select New from the Project menu. Make sure that the location is correct. Create a project called `empty` and add a file called `play.cpp`. Now copy the following code into it.

```
// written by FG on 21/07/04
#include "playpen.h"
#include <iostream>
#include <ostream>

int main(){
 fgw::playpen blank;
 std::cout << "Please press the 'RETURN' key";
 std::cin.get();
}
```

Try to compile the source code file. It will fail with four or more error messages. Here is the failure when I try it on my system:



Only the first error message is important here. The rest are a consequence of that failure to find `playpen.h`. The compiler needs to know where that file is. We could find it ourselves and then copy it into the project directory. In other words, we could put it where the compiler must find it. However, that is a workaround rather than a correct solution. We could also place the complete path inside the quotes for the #include. For example, on my system, I can get the file to compile by replacing that first **`#include`** with:

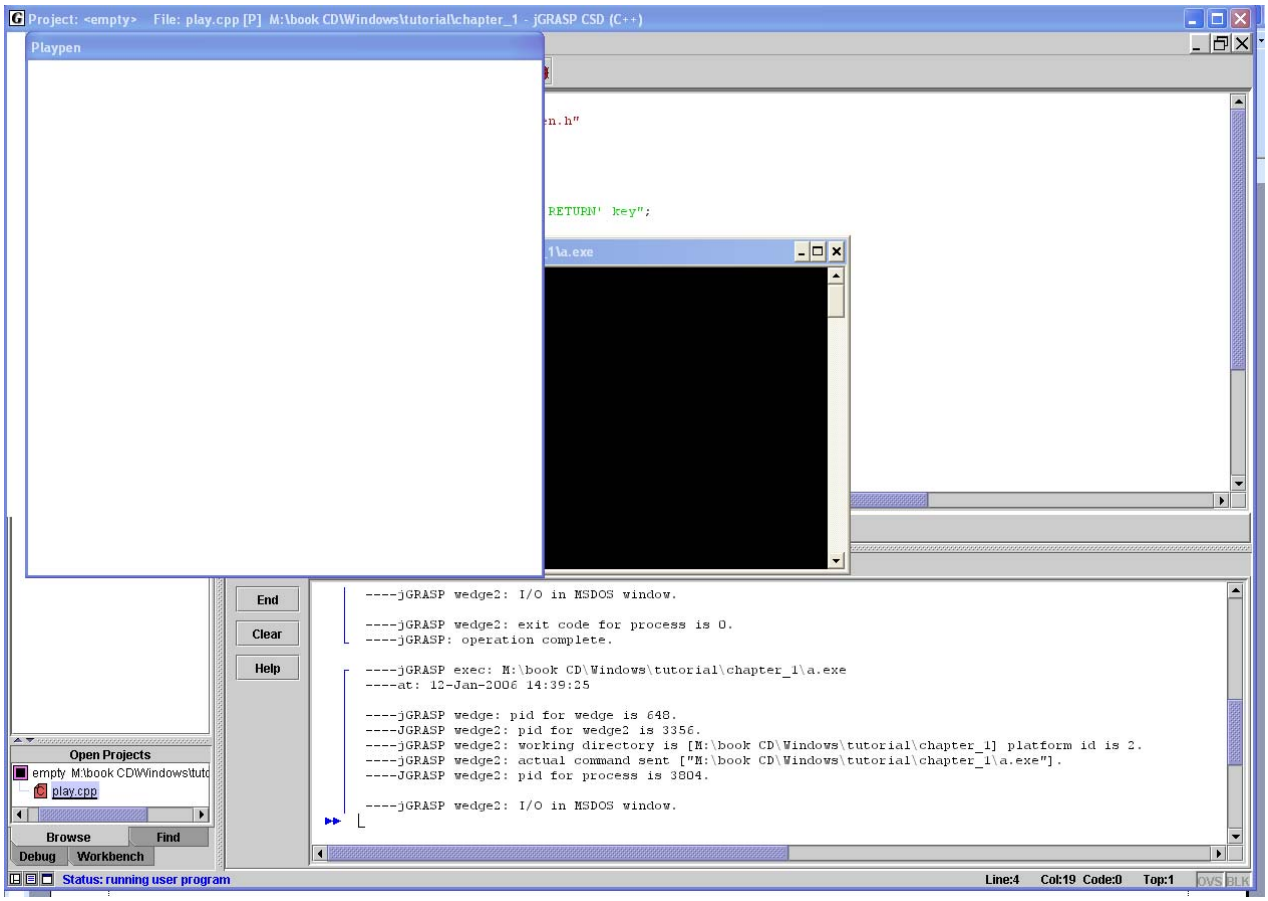
`#include "C:\tutorial\fgw_headers\playpen.h"`

If that were the only problem, I might well do just that (and it is worth knowing this solution for other cases where you just need to tell the compiler where a single header file can be found. Actually, that solution is useful when we just want to compile a file of source code rather than create a program.) However we are going to need to use a `makefile` to deal with the third-party libraries that the linker will need so we might as well choose that route to solve the immediate problem (of getting the code to compile).

Find the version of `makefile4playpen` for your operating system. Make sure that the lines that define C_INCLUDE_DIRS and LIB_DIRS give the correct location of the header files and library files that you installed from the CD. I have set the rest of it to be correct as long as you have used the names I have used in this text.

Now save the result as `makefile` in the current project directory. You may wish first to resave the `makefile` for the previous project with a name such as `makefilem` so that you can recover it later if you wish to.

Now you should be able to make the project and run the result. When you do so, you should see something such as:



Note that this program creates two windows; the currently empty Playpen window on top and the (black) console window that handles user interaction from the keyboard. When you want to interact (e.g. end the program), click on the console window to bring it to the top.

What the Code Means

The first line after the comment is a different form of **`#include`**. The use of quotation marks tells the compiler that this is a header file (as opposed to a system or language header) and that it should look for it in the places designated for such files. By default, this is the same directory as the current file. As we need the compiler to find it elsewhere we have provided the relevant path in the C_INCLUDE_DIRS line of the `makefile`.

The next three lines are the same as for our first program, with the same significance. The line **fgw::playpen blank;** tells the compiler that at this stage in the program you want a Playpen and that you are naming the object representing it **blank**. If you are familiar with the concept of declarations and types, **fgw::playpen** is a type and the whole statement is a declaration of **blank**. I will be covering the details of declarations in the next chapter.

I hope that the meaning of **std::cout << "Please press the 'ENTER' key";** is obvious. The last statement of the program, **std::cin.get( );** may seem strange to some readers. **std::cin** is the standard C++ console input object (the input equivalent of **std::cout**). We largely use it to obtain information from the keyboard. The remainder of the statement tells the compiler that you want to get a single character from the keyboard. C++ only extracts data from the keyboard when signaled that input is complete (usually by the user of the program pressing the Enter key).

We have to use some way to keep the program running until we have finished with the Playpen, because the window will close automatically when the program ends. Try removing the **std::cin** statement (or commenting it out by inserting **//** at the beginning of the line) and then running the revised program. The Playpen just flashes on the screen. The **std::cin.get( )** causes the program to wait until the user provides some input by pressing Enter.

Something to Play With

None of the three programs you have tried does anything even vaguely interesting but if you would like to try something a little more exciting you can read Chapter 1 of *You Can Do It!* (on the CD), which will provide you with more information about what can be done in a Playpen window.

Summary

JGrasp™ is an excellent IDE designed for learning. However one of the costs of its versatility is that it makes somewhat heavier demands on the isolated user. In an academic environment, a teaching assistant or another student will normally be on hand to explain and help with problems. The need to use **`make`** for more advanced projects is a barrier to those with little prior experience but for those familiar with the concept it should actually increase the power. For others who intend to continue into a professional programming environment, acquiring some facility with **`make`** will be an advantage in the long run.

Every C++ program includes a function called **`main()`** which is used as the entry point for the program.

C++ includes a pair of objects, **`std::cout`** and **`std::cin`** that represent the console output device (defaulting to a window on the monitor) and console input device (defaulting to the keyboard). The **`iostream`** header makes these objects available.

We can send (shift out) data to the **`std::cout`** by using **`<<`** and we can extract a single character from **`std::cin`** by using the **`get()`** function.